Оценка движения (Motion Estimation)

Оценка движения – один из основных алгоритмов, применяемых при обработке и сжатии видеоданных в кодеке H264. Алгоритм использует схожесть соседних кадров в видео последовательности и находит векторы движения отдельных частей изображения (обычно блоков 16×16 и 8×8). Использование компенсации позволяет при сжатии многократно увеличить степень сжатия за счёт удаления избыточности в виде совпадающих частей кадров. Используется не только при сжатии, но и при фильтрации видео, изменении частоты кадров и т.д.

Практически в любом видео соседние кадры похожи, имеют общие объекты, которые, как правило, смещаются друг относительно друга. И совершенно естественно желание закодировать видео так, чтобы объекты не кодировались многократно, а просто описывались некоторые их смещения.

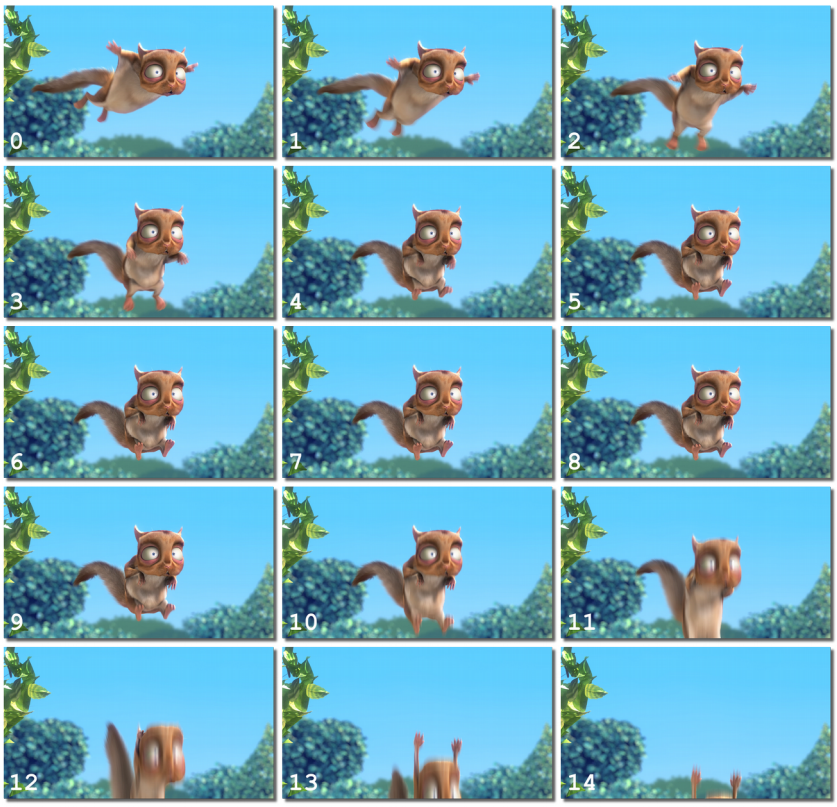

Рисунок 1: Тестовая последовательность

На рисунке 1 явна видна схожесть соседних кадров, что типично для любого видео.

Пример работы алгоритма

В первую очередь изображение делится на блоки:

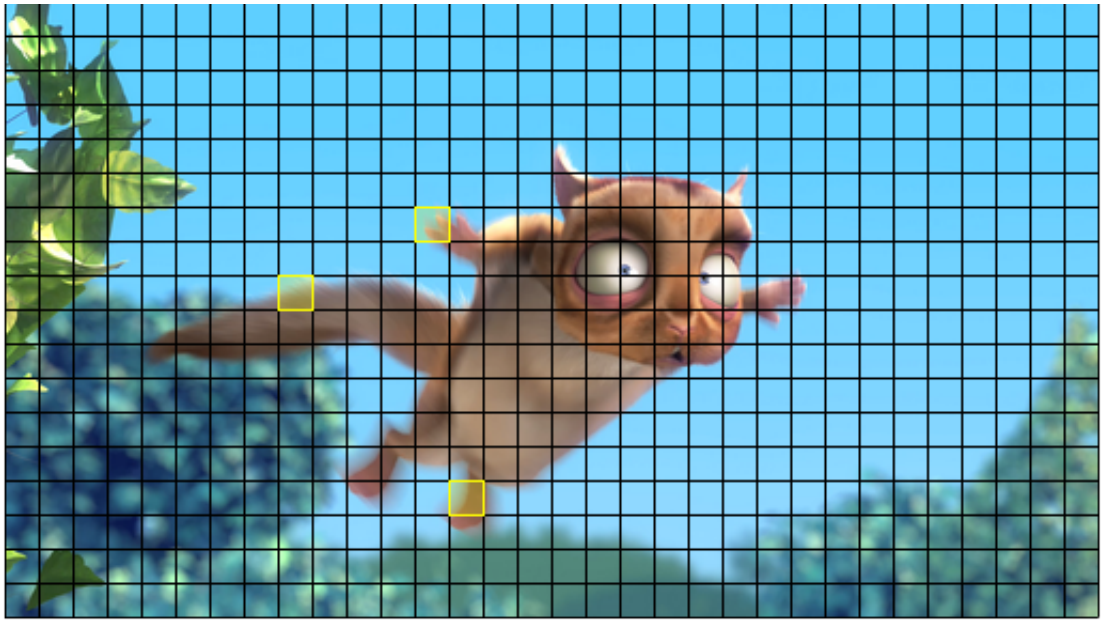

Рисунок 2: Деление изображения на блоки

Далее происходит поиск наиболее подходящего блока в соседнем кадре:

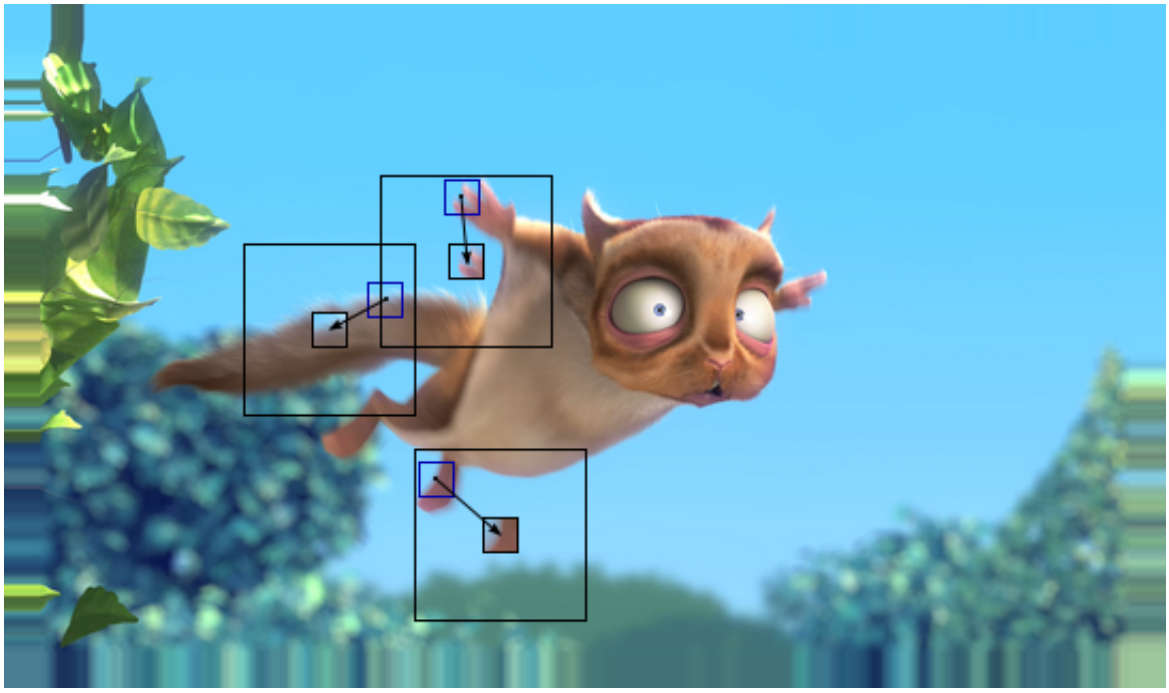

Рисунок 3: Поиск совпадающих блоков

Для нахождения наиболее коррелируемых блоков можно вычислять SSD (суммы квадратичных отклонений) или SAD (суммы абсолютных разностей). SAD имеет меньшую устойчивость к шуму, по сравнению с SSD, но будет использоваться в виду хороших скоростных показателей за счет использования SIMD (одна инструкция, много данных).

Three Step Search (Трехшаговый поиск)

В виду того, что алгоритм оценки движения является наиболее затратным по времени, из за необходимости большого количества расчетов (много раз считается SAD), необходимо использовать оптимальный алгоритм поиска блоков. Алгоритм Three Step Search (TSS) не самый лучший алгоритм на данный момент, но так как отличается не количественно, а качественно (устойчивость к шуму и т.д.), то будет применен именно он.

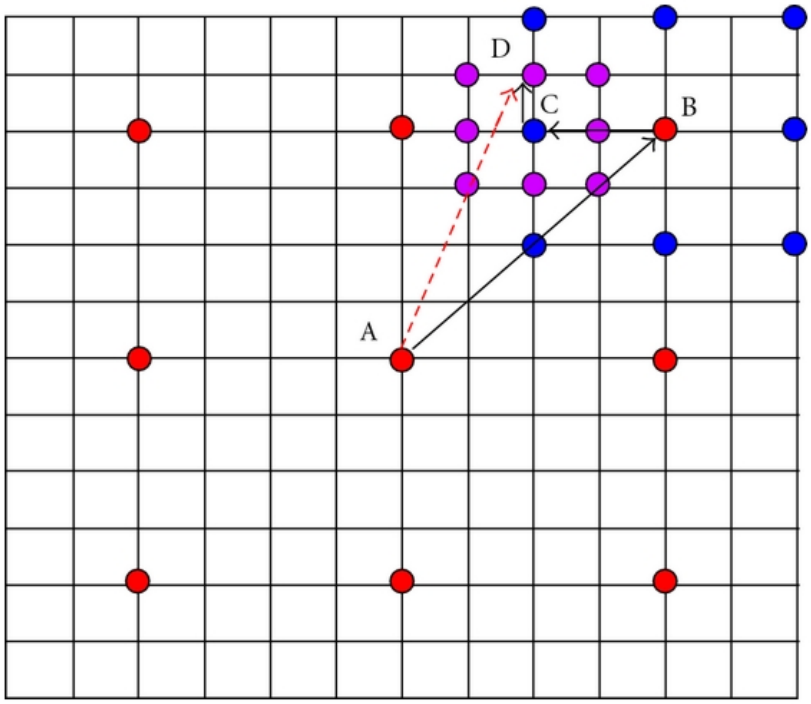

Рисунок 4: Пример алгоритма TSS

В общих словах смысл алгоритма TSS таков: устанавливается шаг поиска равный 4 пикселям (для области поиска p = 7) и вычисляется SAD для девяти блоков, далее центр переносится в область с наименьшим значением SAD и поиск продолжается дальше, с шагом, разделенным на 2.

Сложность алгоритма: O(8*log(p)) = O(8*log(r/2)), где r – зона поиска (в пикселях), p – размер начального шага (в пикселях).

Основные операции: $\Sigma \left| X_i - Y_i \right|$,

Реализация алгоритма

В независимости от того применяется SIMD или нет общий алгоритм остается неизменным. Расширение SIMD будет применятся только на этапе вычисления SAD.

1) Не будем учитывать вид, в котором представлены входные данные и воспользуемся «bitmap_image.hpp». (*github.com/ArashPartow/bitmap/blob/master/bitmap_image.hpp*).

Фрагмент кода функции, вычисляющей SAD:

previous_frame.get_pixel(i, j, colour_prev);

current_frame.get_pixel(i1, j1, colour_cur);

red_prev[(j-y1)%4] += int(colour_prev.red);

green_prev[(j-y1)%4] += int(colour_prev.green);

blue_prev[(j-y1)%4] += int(colour_prev.blue);

red_curr[(j-y1)%4] += int(colour_cur.red);

green_curr[(j-y1)%4] += int(colour_cur.green);

blue_curr[(j-y1)%4] += int(colour_cur.blue);

red_error = __builtin_msa_asub_s_w(red_prev, red_cur);

green_error = __builtin_msa_asub_s_w(green_prev, green_cur);

blue_error = __builtin_msa_asub_s_w(blue_prev, blue_cur);

Как видно, чтобы использовать инструкцию вычитания векторов MSA необходимо поэлементно инициализировать массив MSA, что не является рациональным решением. Сравним общее количество инструкций обычного алгоритма и алгоритма с использованием MSA (с поэлементной инициализацией):

Обычная реализация *total: 8020188*

Реализация с MSA

Название инструкции	**Количество вызовов**
ADDV.df	200
ASUB_S.df	200
LD.df	1875
LDI.df	468
ST.df	870
OPC_COPL_LDST	221
OPC_LD	2966844
OPC_ARITH_IMM	1466630
OPC_SHIFT_IMM	553673
OPC_SLT_IMM	4172
OPC_LOGIC	1765728
OPC_LOGIC_IMM	200894
OPC_COND_MOVE	7620
OPC_ST	1872298

OPC_SHIFT	105
OPC_ST_COND	103
OPC_FLT_LDST	221
OPC_SLT	5312
OPC_ARITH	895976

Total: 9743410; Процент MSA: 0.04%

Как видно такая реализация с MSA не уменьшает количество вызываемых команд, а увеличивает, поэтому такой подход не применим.

2) Входные данные в виде массива

Проверим какие инструкции вызываются при копировании обычного массива в массив MSA.

v4u32 *a;*

int *b[4] = {1, 2, 3, 4};*

*a = *(v4u32*)b;*

Помимо обычных инструкций вызвались следующие:

LD.df: 1 — Vector load

ST.df: 1 — Vector store

В связи с этим входные данные необходимо представлять в виде массива, чтобы реализация на MSA имела смысл.

2.1) Выбор типа данных

В MSA существует 8 типов массивов с целочисленными значениями: v16u8, v16i8, v8u16, v8i16, v4u32, v4i32, v2u64, v2i64. Необходимо выяснить какой из типов нам подойдет лучше.

Беззнаковые типы нам не подходят, так как необходимо считать разность между пикселями (которая может быть отрицательной). Для корректной работы алгоритма для каждого канала пикселя необходимо выделить свой массив.

Диапазон значений массива v16i8 - [-128; 127] => этот массив нельзя использовать, в виду того, что максимальное значение пикселя — 255.

Рассмотрим массив v8i16. Этот массив имеет 8 16-ти битных значений и его диапазон - [-32768, 32767]. Максимальная сумма при накапливании значений каналов пикселей для блоков 16x16 — 16*16*255 = 65280. Так как у массива 8 значений, то максимальное значение каждого элемента массива — 65280/8 = 8160. Так как данный массив избыточен для блока, с размером 16x16 массивы v4i32 и v2i64 рассматривать не будем.

Можно заметить, что при использовании блока, с размером 32x32 максимальная сумма при накапливании значений — 32*32*255 = 261120. 261120/8 = 32640 — это значение принадлежит диапазону значений вектора v8i16, поэтому можно использовать блоки 32x32 не изменяя тип массива MSA.

2.2) Профилирование алгоритма

Фрагмент кода функции, вычисляющей SAD:

red_prev+=((v8i16*)(red_prevv+(i-x1)));*

green_prev+=((v8i16*)(green_prevv+(i-x1)));*

blue_prev+=((v8i16*)(blue_prevv+(i-x1)));*

red_cur+=((v8i16*)(red_curr+(i-x1)));*

green_cur+=((v8i16*)(green_curr+(i-x1)));*

blue_cur+=((v8i16*)(blue_curr+(i-x1)));*

Где выделенные зеленым цветом переменные — массивы входных данных.

Название инструкции	**Количество вызовов**
ADDV.df	13064
ASUB_S.df	200
LD.df	29144
ST.df	13734
OPC_ARITH	30513
OPC_SLT	2
OPC_ST	5452
OPC_COPL_LDST	199
OPC_SHIFT_IMM	14281
OPC_COND_MOVE	2
OPC_LOGIC	4138
OPC_ST_COND	4
OPC_SHIFT	1
OPC_LOGIC_IMM	291
OPC_LD	59001
OPC_FLT_LDST	199
OPC_ARITH_IMM	5433
OPC_SLT_IMM	3

Total: 175661; Процент MSA: 32%

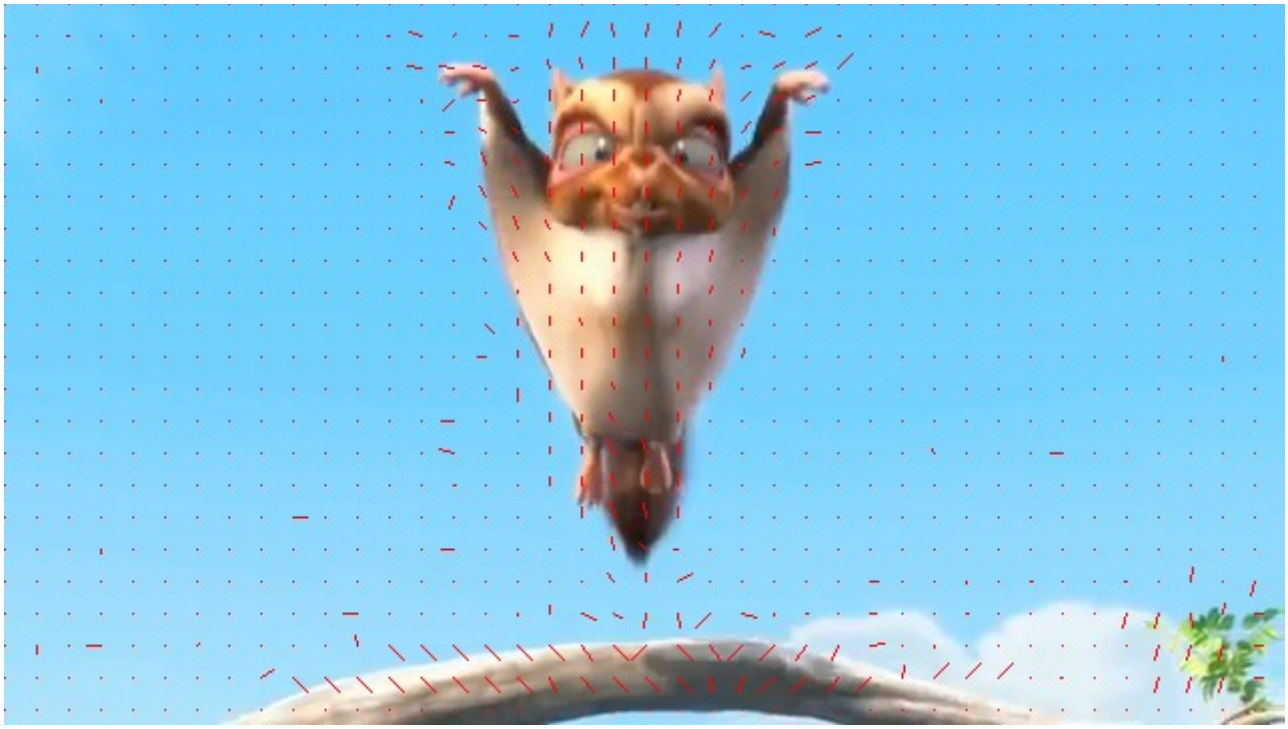

Рисунок 5: Результат работы алгоритма

Рекомендации:

Введение инструкции, которая позволяет вычитать из беззнакового массива беззнаковый и получать либо знаковый массив, либо абсолюную разницу, позволит использовать вектор v16u8, который может лучше подходить для алгоритма.